Deep Neural Networks

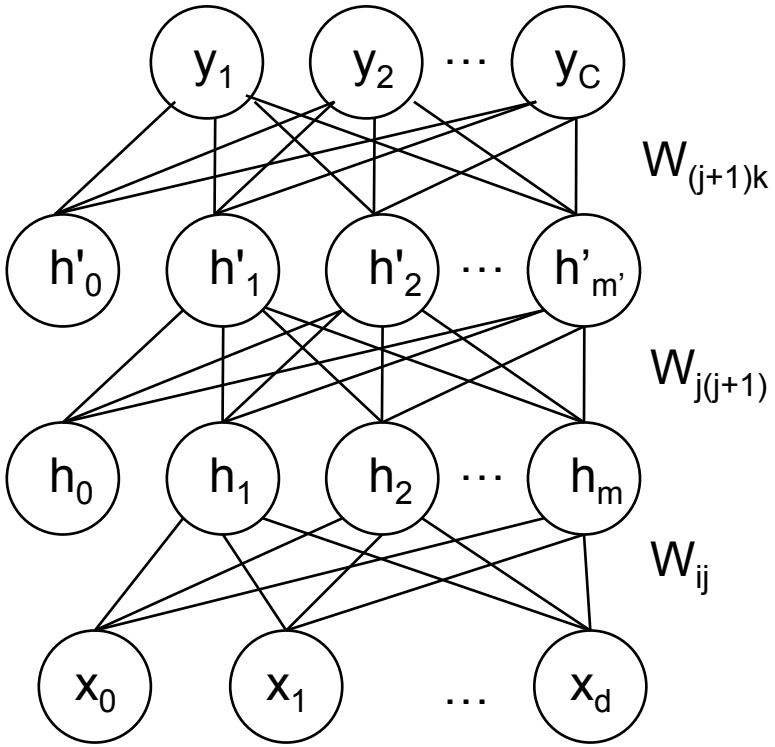
CS 550: Machine Learning

Deep Architectures

- They are hard to train by backpropagation due to the vanishing gradient problem

$$\frac{\partial loss}{\partial W_{ij}} = \frac{\partial loss}{\partial net_j} \frac{\partial net_j}{\partial W_{ij}} = \delta_j x_i$$

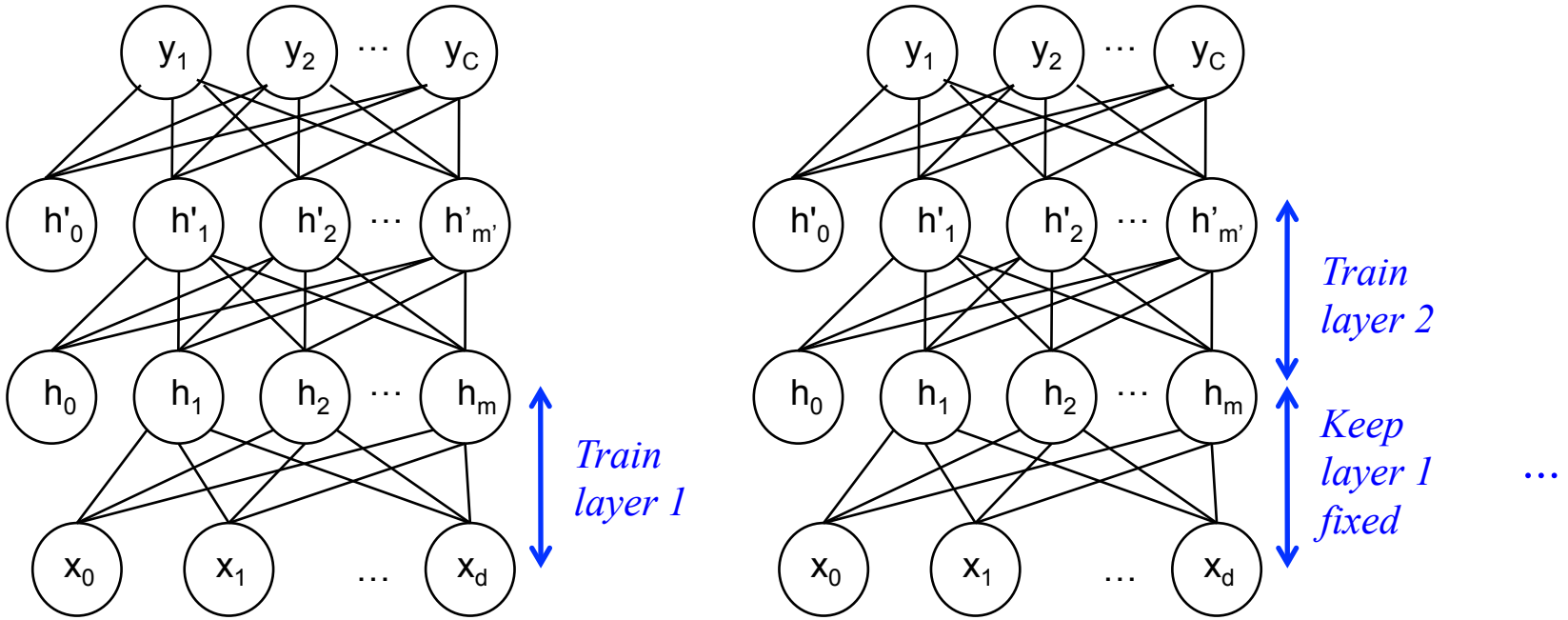
$$\delta_j = \left[\sum_{(j+1)} \delta_{(j+1)} W_{j(j+1)} \right] \sigma'(net_j)$$



- However, when the initial weights are good enough, backpropagation works well
- Layerwise pretraining
  - Restricted Boltzmann machines
  - Autoencoders

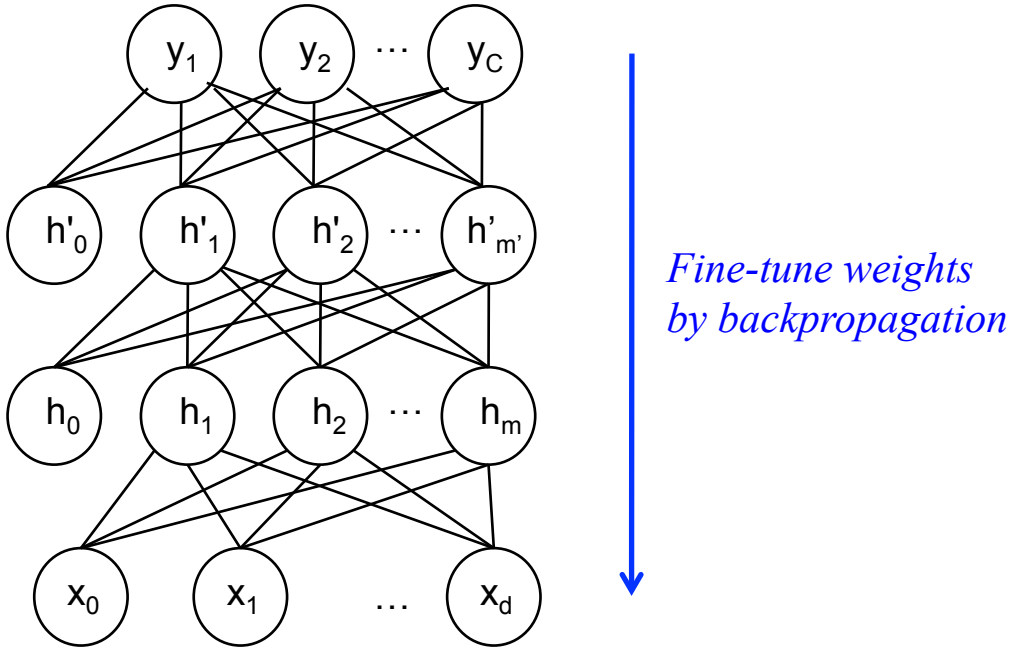
Layerwise Pretraining

- First, train one layer at a time, optimizing P(x)



Layerwise Pretraining

- Then, fine-tune weights, optimizing P(y|x) by backpropagation



Restricted Boltzmann Machines (RBMs)

- RBM is a simple energy-based model

$$p(x, h) = \frac{1}{Z_\theta} \exp(-E_\theta(x, h))$$

$$E_\theta(x, h) = -x^\top W h - b^\top x - d^\top h$$

$$Z_\theta = \sum_{(x, h)} \exp(-E_\theta(x, h))$$

normalizer

It only allows h-x interactions

- Stacked RBMs can be used to construct a deep belief net, which is a probabilistic generative model
- Stacked RBMs can also be used to initialize a deep neural network

Training RBMs to optimize P(x)

Maximize the log-likelihood of data

$$\underbrace{\partial_{W_{ij}} \log P_W(x = x^{(m)})}_{\text{Derivative of the log-likelihood}} = \partial_{W_{ij}} \log \sum_h P_W(x = x^{(m)}, h)$$

$$= -\partial_{W_{ij}} \log Z_W + \partial_{W_{ij}} \log \sum_h \exp(-E_W(x^{(m)}, h))$$

$$= \underbrace{-\mathrm{E}_{p(x,h)}[x_i\ h_j]}_{\text{Negative phase comes from the model}} + \underbrace{\mathrm{E}_{p(h \mid x = x^{(m)})}[x_i^{(m)}\ h_j]}_{\text{Positive phase comes from the data}}$$

The negative phase term is expensive to calculate since it requires sampling (x, h) from the model. **Contrastive divergence** is a faster solution.

Contrastive Divergence Algorithm

Initialize with the training sample and wait only a few sampling steps (usually 1 step)

1. Let $x^{(m)}$ be training sample and w_{ij} , b_i , d_j be current weights
2. Sample $\hat{h}_j \in \{0 , 1\}$ from $p(h_j \mid x = x^{(m)}) = \sigma\left(\sum_i w_{ij} x_i^{(m)} + d_j\right)$, $\forall j$
3. Sample $\tilde{x}_i \in \{0 , 1\}$ from $p(x_i \mid h = \hat{h}) = \sigma\left(\sum_j w_{ij} \hat{h}_j + b_i\right)$, $\forall i$
4. Sample $\tilde{h}_j \in \{0 , 1\}$ from $p(h_j \mid x = \tilde{x}) = \sigma\left(\sum_i w_{ij} \tilde{x}_i + d_j\right)$, $\forall j$

h's and x's are assumed to be binary variables

$$
\begin{aligned}
w_{ij} &= w_{ij} + \gamma\left(x_i^{(m)} \hat{h}_j - \tilde{x}_i \tilde{h}_j\right) \\
b_i &= b_i + \gamma\left(x_i^{(m)} - \tilde{x}_i\right) \\
d_j &= d_j + \gamma\left(\hat{h}_j - \tilde{h}_j\right)
\end{aligned}
$$

Autoencoders

- They learn to "compress" and "reconstruct" input data

Encoder: $h = \sigma(Wx + b)$

Decoder: $x' = \sigma(W'h + d)$

- Learn the weights to minimize the reconstruction loss

$$loss = \sum_m \left(x^{(m)} - x' \right)^2$$

- This is the same backpropagation for a network with one hidden layer, where $x^{(m)}$ is both input and output
- They can be stacked to form a deep neural network
  - Cheaper alternatives to RBMs
  - However, unlike RBMs, they are deterministic and cannot form a deep generative model

Denoising Autoencoders

- Perturb input sample by adding noise to it

Encoder: $h = \sigma(W\tilde{x} + b)$

$\tilde{x} = x + noise$

Decoder: $x' = \sigma(W' h + d)$

- Learn the weights to minimize the reconstruction loss with respect to the original input sample

$$loss = \sum_m \left(x^{(m)} - x' \right)^2$$

Layerwise Pretraining

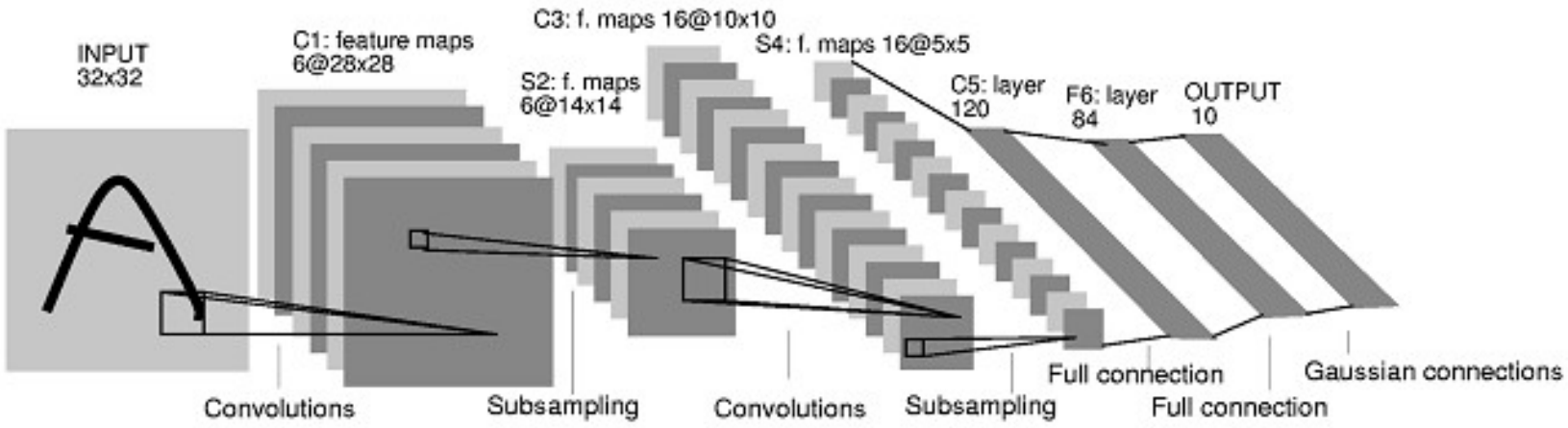
Is it always necessary?

- Answer in 2006: Yes!
- Answer in 2014: No!
  - If initialization is done well by design (e.g., sparse connections and convolutional nets), there may not a vanishing gradient problem
  - If the net is trained on an extremely large dataset, it may not overfit

Slide credit: K. Duh

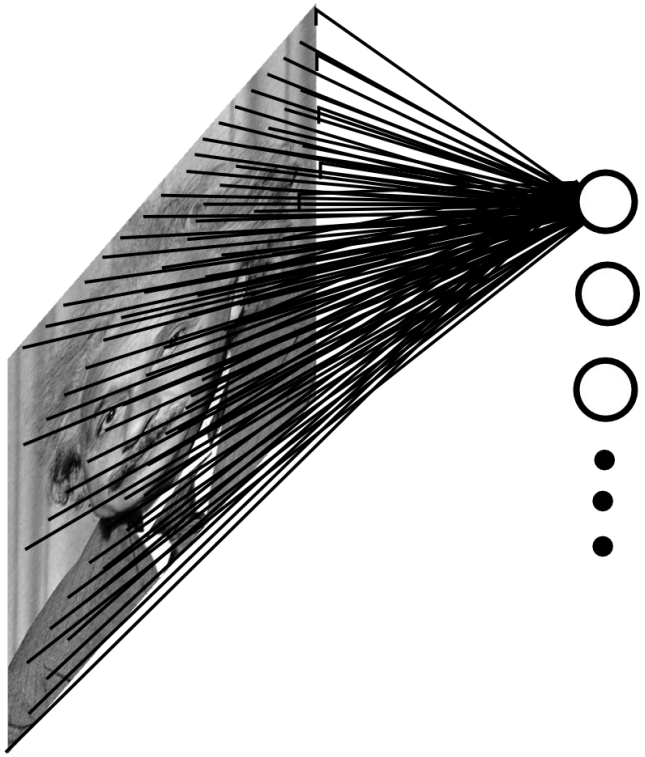
Convolutional Neural Networks

- A CNN consists of a number of convolutional and subsampling (pooling) layers optionally followed by fully connected layers



Convolutional Neural Networks

- When the input data is an image, a fully connected layer will produce a huge number of weights (parameters) to be learned

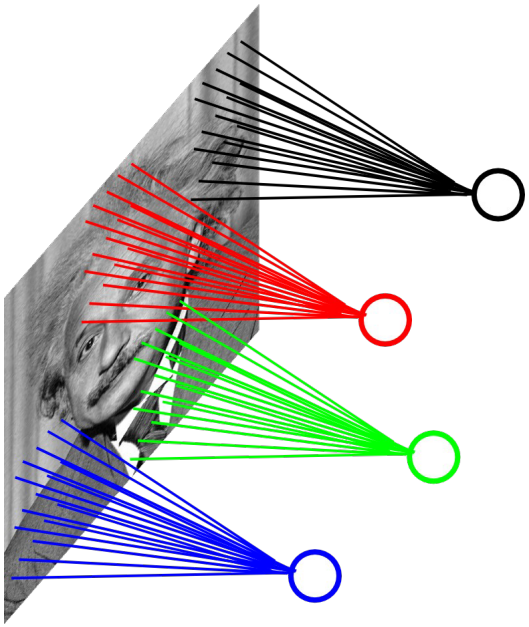


Example:
200x200 image
25K hidden units
➔ ~1B parameters

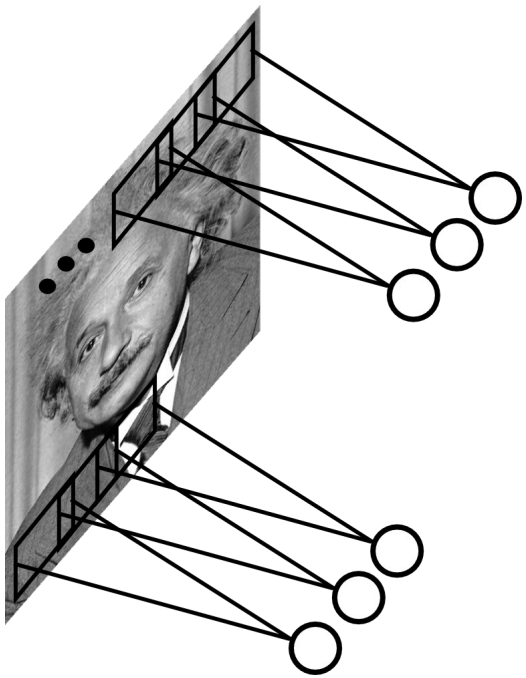
Slide credit: M. A. Ranzato

Convolutional Layer

- However, spatial correlation is local and statistics is similar at different locations
- Thus, small kernels are defined and their parameters are shared by all pixels
- **It is convolution with learned kernels**

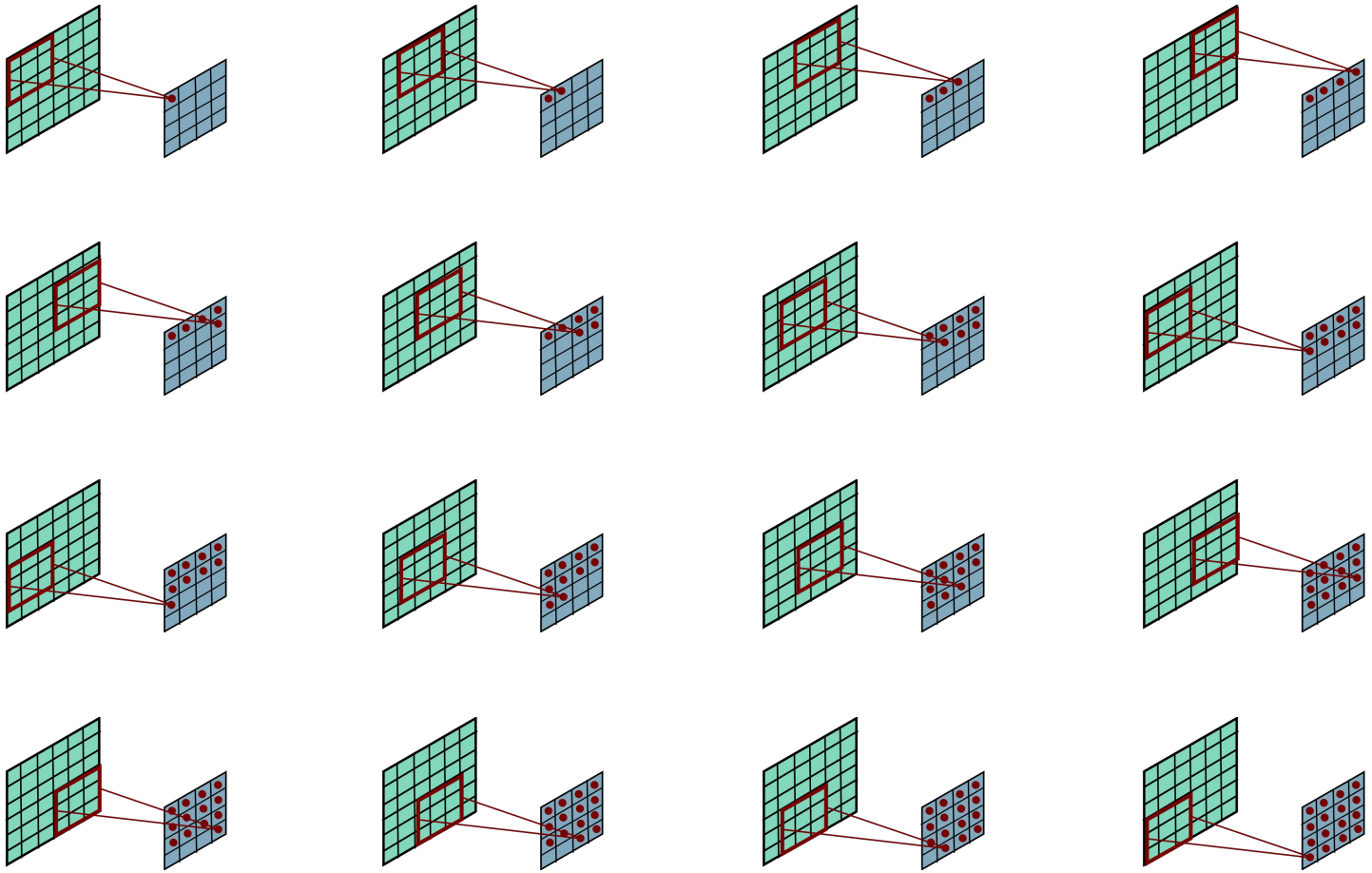


Example:
200x200 image
25K hidden units
10x10 kernels
➔ ~2.5M parameters
(instead of 1B parameters)



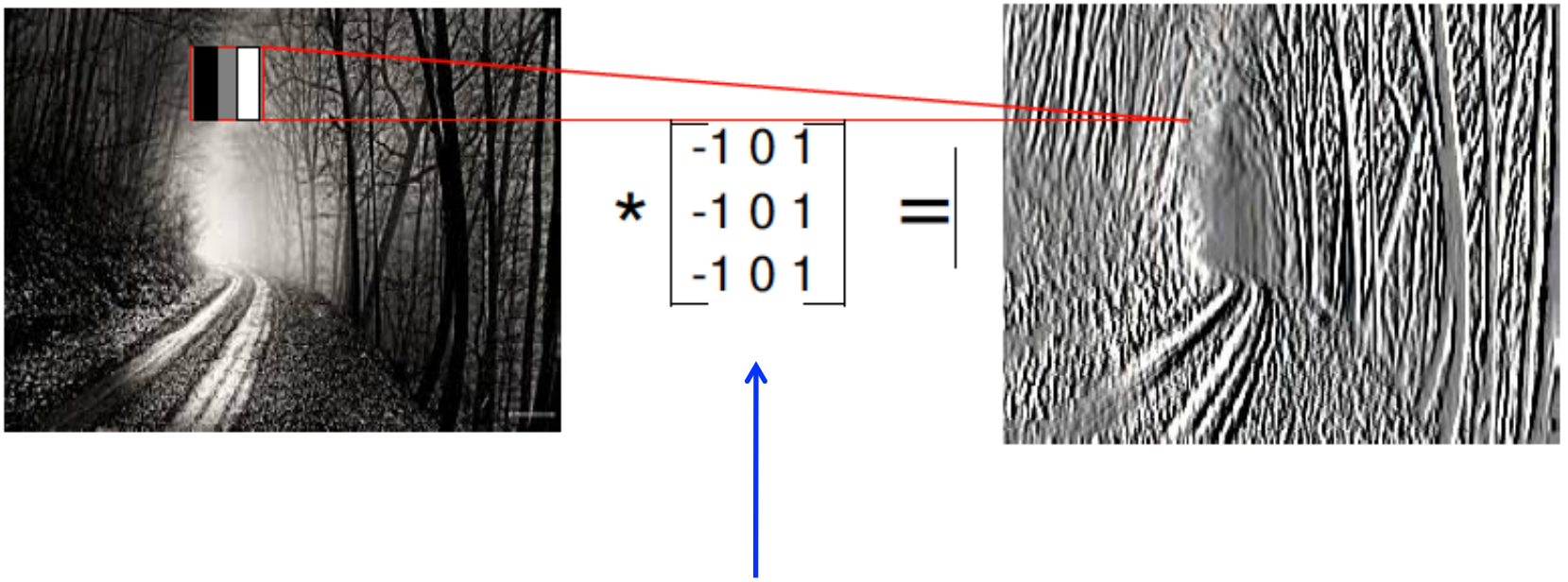
Slide credit: M. A. Ranzato

Convolutional Layer



Slide credit: M. A. Ranzato

Convolutional Layer

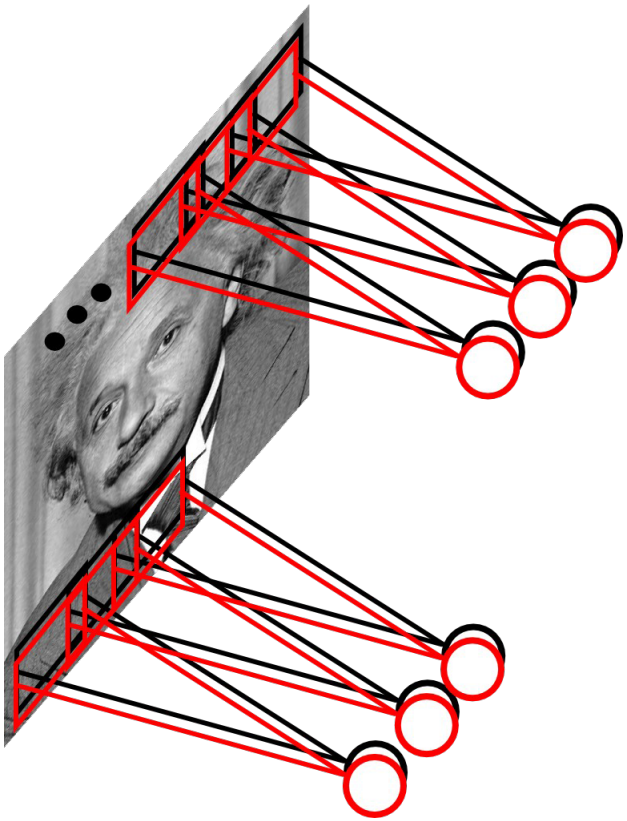


Learn this filter from data!!!

Slide credit: M. A. Ranzato

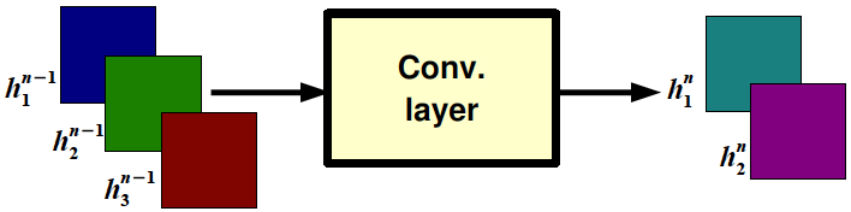
Convolutional Layer

Learn multiple filters



$$h_j^n = \max\left(0\,,\ \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map — *input feature map* — *kernel*



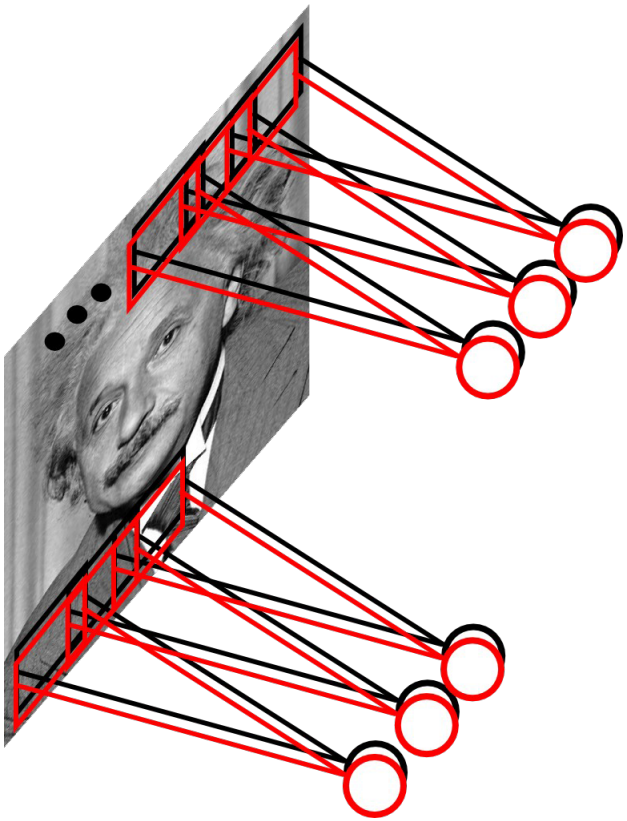
Rectified linear unit (ReLU) provides nonlinearity: u = max(0, x)

- fast to compute
- reduces the likelihood of the gradient to vanish
- better sparsity

Slide credit: M. A. Ranzato

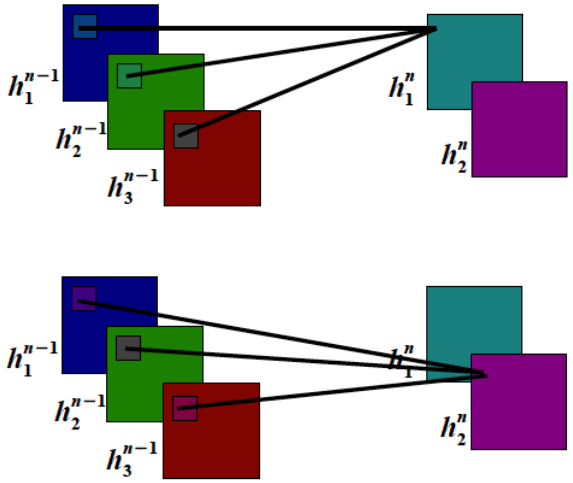
Convolutional Layer

Learn multiple filters



$$h_j^n = \max\left(0\,,\ \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

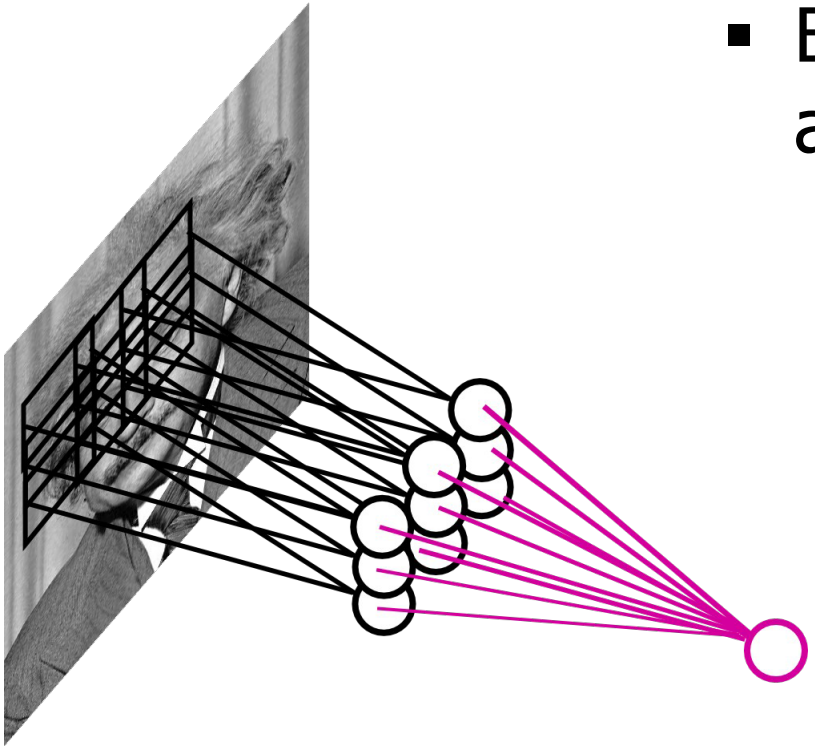
output feature map — *input feature map* — *kernel*



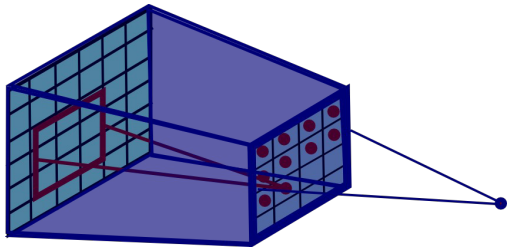
Choosing the architecture (the number of feature maps, size of kernels, and number of convolutional layers) is task dependent

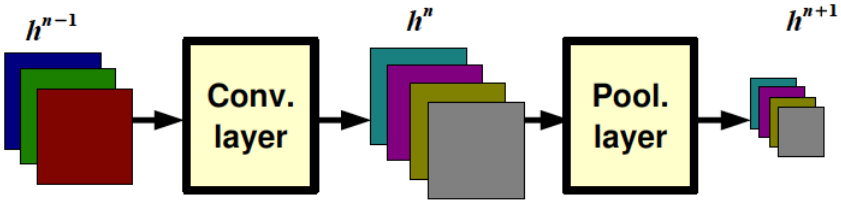
Slide credit: M. A. Ranzato

Pooling Layer



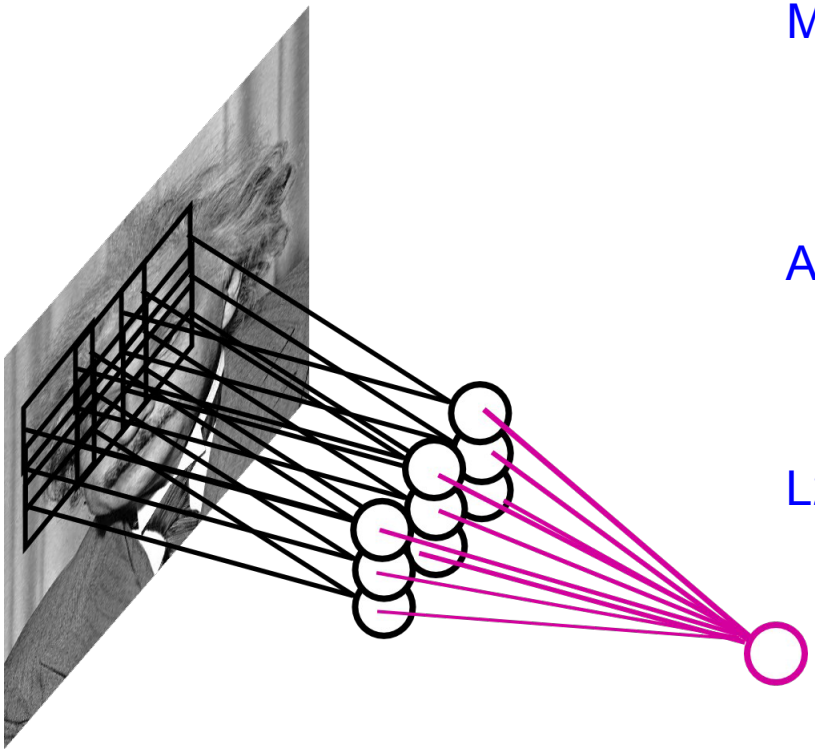
- By pooling filter responses at different locations
  - We gain robustness to the exact location of features
  - Receptive field becomes larger for the next layer (the next layer will look at larger spatial regions)





Slide credit: M. A. Ranzato

Pooling Layer



Max-pooling:

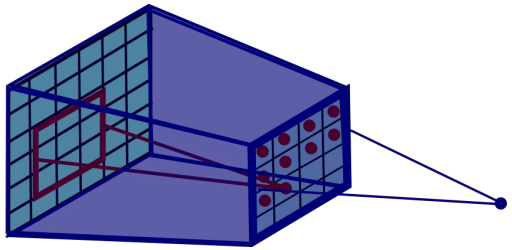
$$h_j^n(x, y) = \max_{\substack{\bar{x} \in N(x) \\ \bar{y} \in N(y)}} h_j^{n-1}(\bar{x}, \bar{y})$$

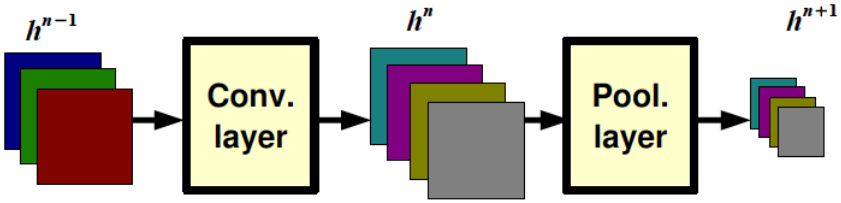
Average-pooling:

$$h_j^n(x, y) = 1/K \sum_{\substack{\bar{x} \in N(x) \\ \bar{y} \in N(y)}} h_j^{n-1}(\bar{x}, \bar{y})$$

L2-pooling:

$$h_j^n(x, y) = \sqrt{\sum_{\substack{\bar{x} \in N(x) \\ \bar{y} \in N(y)}} h_j^{n-1}(\bar{x}, \bar{y})^2}$$



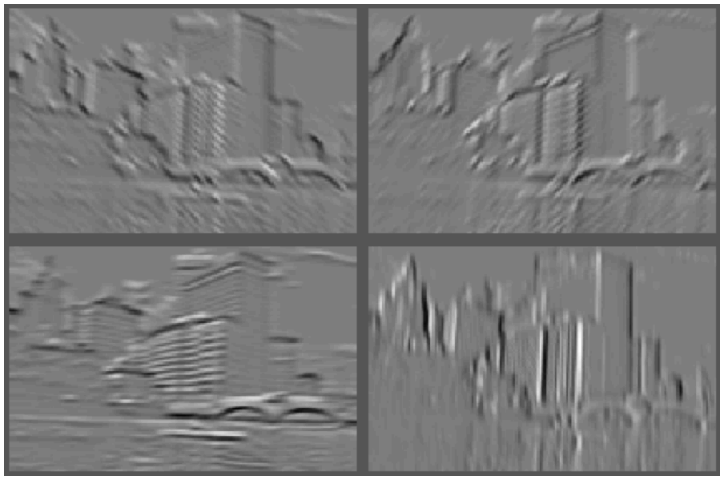


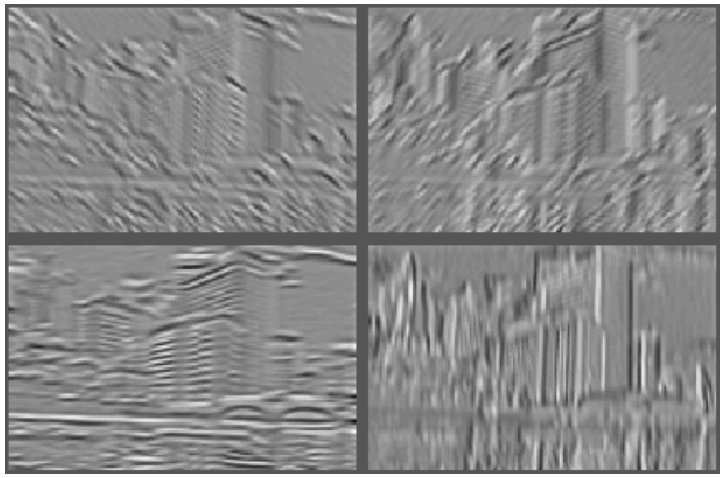
Slide credit: M. A. Ranzato

Local Contrast Normalization

- Equalizes feature maps/responses

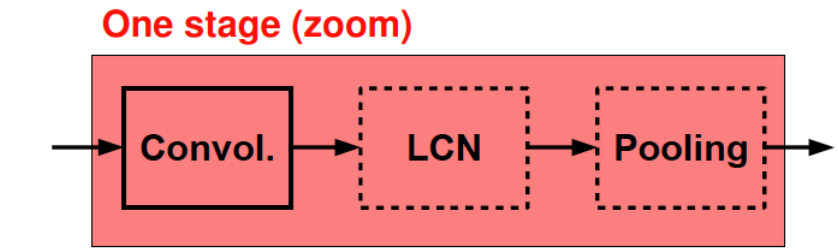
$$h^{n+1}(x\,,\,y) \;=\; \frac{h^{n}(x\,,\,y) \;-\; m^{n}(N(x\,,\,y))}{\max(\varepsilon\,,\,\sigma^{n}(N(x\,,\,y)))}$$

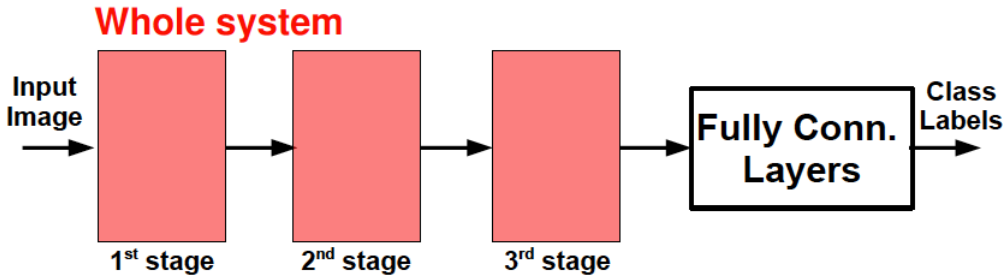




Slide credits: R. Fergus and M. A. Ranzato

CNNs: Typical Architecture





All layers are differentiable so that standard backpropagation can be used

After one stage

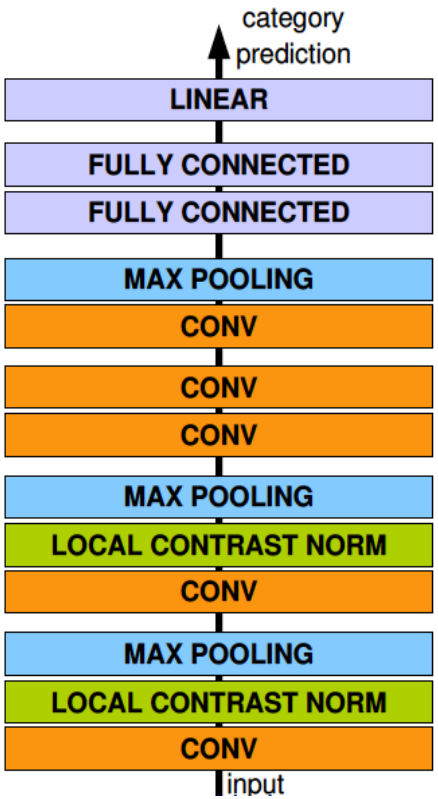
- Number of feature maps is usually increased (conv. layer)
- Spatial resolution is usually decreased (pooling layer and stride in conv. layer)
- Receptive field gets larger

After several stages

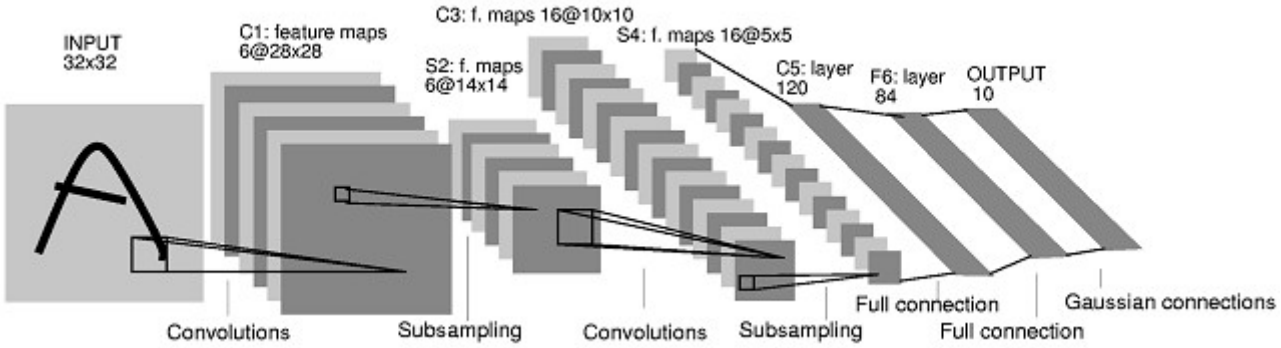
- Spatial resolution is greatly reduced and number of feature maps is large so convolution would not make any sense
- Next layer(s) will consist of fully connected layers (with or without hidden layers)

Slide credit: M. A. Ranzato

CNN Examples



ImageNet by Krizhevsky et al., 2012

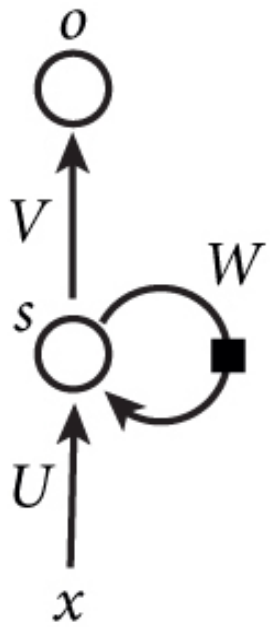


LeNet-5 by LeCun et al., 1998

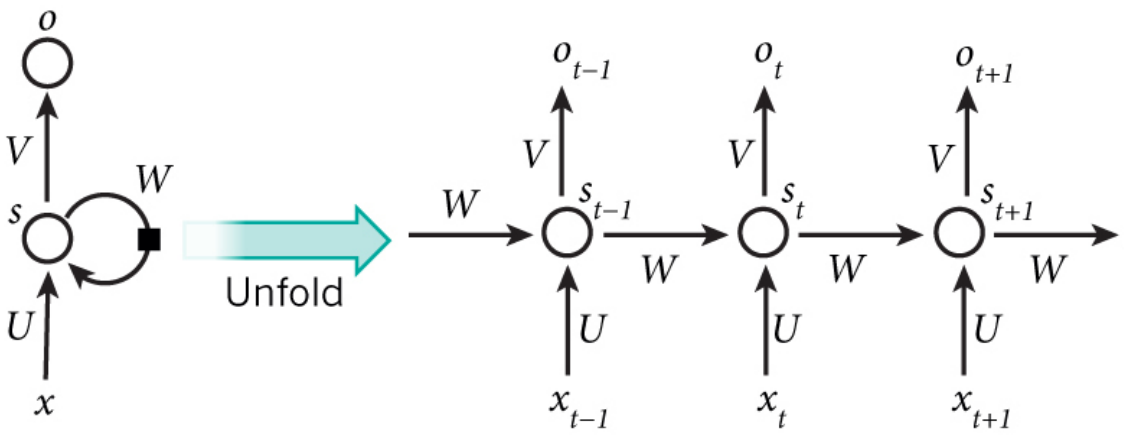
Slide credit: M. A. Ranzato

Recurrent Neural Networks

- Feedforward neural networks assume that all inputs/outputs are independent
  - However, it is not true for sequential data (speech recognition, translation, etc)
- **Recurrent neural networks** do not have this assumption
  - They perform the same task for every element of a sequence, with the output being dependent on previous computations
  - They might be considered to have a "memory" that captures information about what has been calculated so far



Recurrent Neural Networks



RNN unfolded in time

Input at time t

$$x_t$$

Hidden state at time t

$$s_t = f(U x_t + W s_{t-1})$$

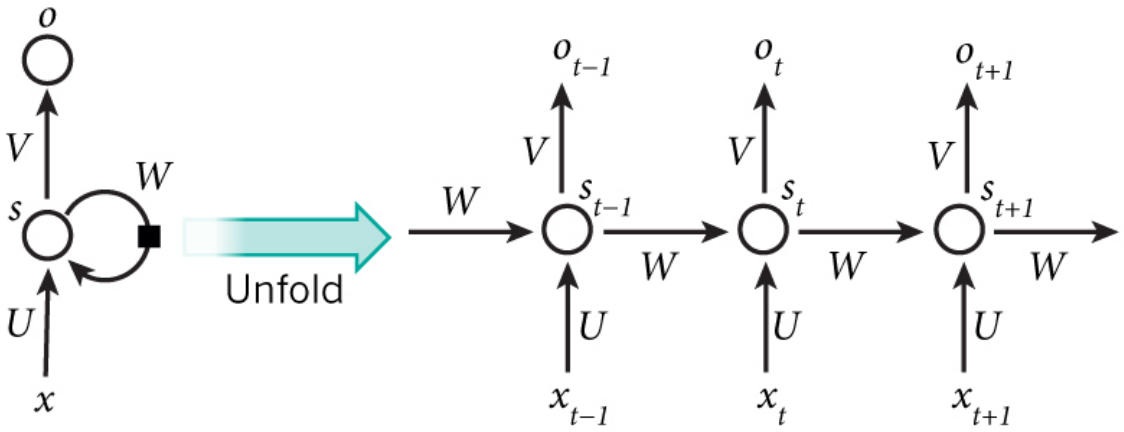
Output at time t

$$o_t = \text{softmax}(V s_t)$$

f is a non-linear function such as tanh or ReLU

- All steps of an RNN share the same weights (U, V, W)
  - This reflects the fact that each step performs the same task just with a different input
  - Unlike a deep neural network (which uses different weights at each different layer)

Recurrent Neural Networks



RNN unfolded in time

Input at time t

$$x_t$$

Hidden state at time t

$$s_t = f(U x_t + W s_{t-1})$$

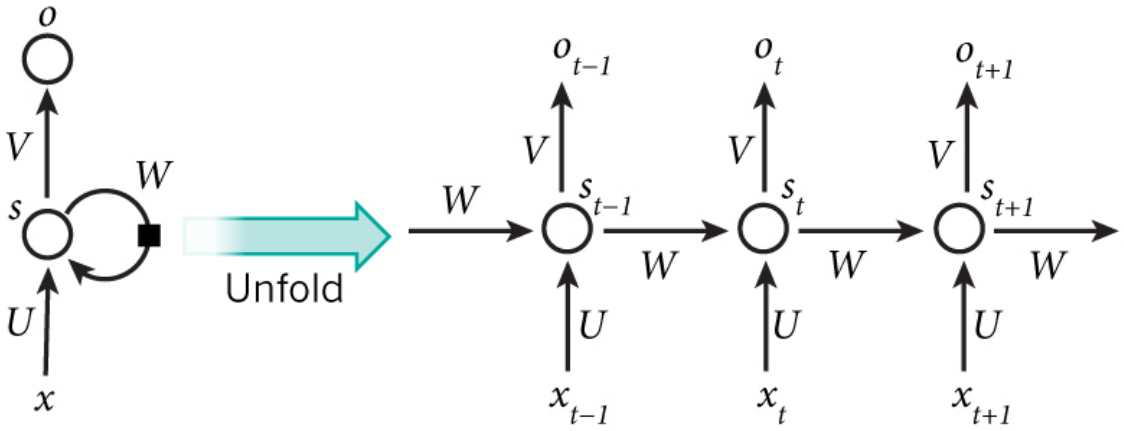
Output at time t

$$o_t = \text{softmax}(V s_t)$$

f is a non-linear function such as tanh or ReLU

- Training an RNN also uses backpropagation (called backpropagation through time – BPTT)
  - In theory, RNNs can learn arbitrarily long sequences
  - But, in practice, they have difficulties in learning long-term dependencies

Training Recurrent Neural Networks



RNNs with many steps are hard to train due to the vanishing gradient problem

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

$$o_t = \text{softmax}(Vs_t)$$

$$E(y, o) = \sum_t E(y_t, o_t)$$

$$E(y, o) = \sum_t \underbrace{-y_t \log o_t}_{\textit{cross entropy}}$$

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

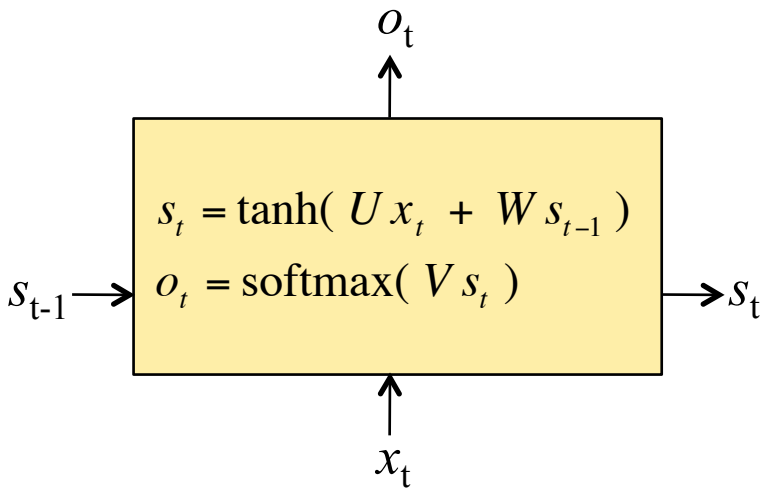
$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial o_t}\frac{\partial o_t}{\partial s_t}\frac{\partial s_t}{\partial W} = \frac{\partial E_t}{\partial o_t}\frac{\partial o_t}{\partial s_t}\sum_{k=0}^{t}\frac{\partial s_t}{\partial s_k}\frac{\partial s_k}{\partial W}$$

$$\frac{\partial s_t}{\partial s_k} = \frac{\partial s_t}{\partial s_{t-1}}\frac{\partial s_{t-1}}{\partial s_{t-2}}\cdots\frac{\partial s_{k+1}}{\partial s_k}$$

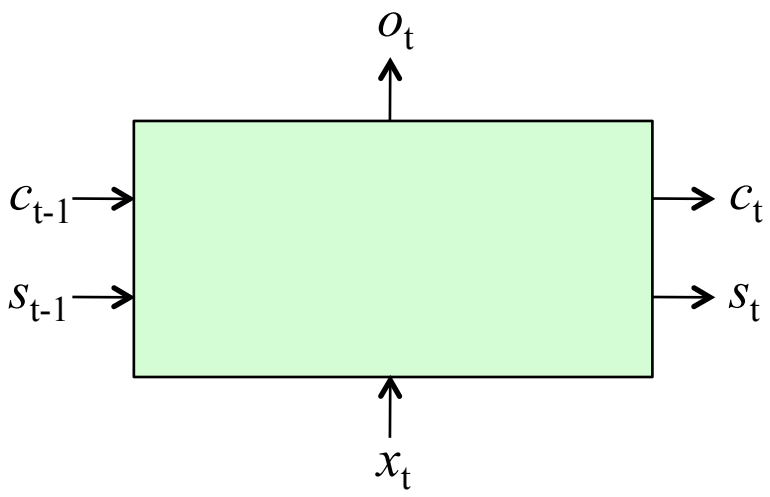
$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial o_t}\frac{\partial o_t}{\partial s_t}\sum_{k=0}^{t}\left(\prod_{j=k+1}^{t}\frac{\partial s_j}{\partial s_{j-1}}\right)\frac{\partial s_k}{\partial W}$$

Long Short Term Memory Networks

- LSTM network is a type of RNN, which is explicitly designed to avoid long-term dependency problem
- It introduces an additional cell state c_t that controls the flow of information over time



Standard RNN

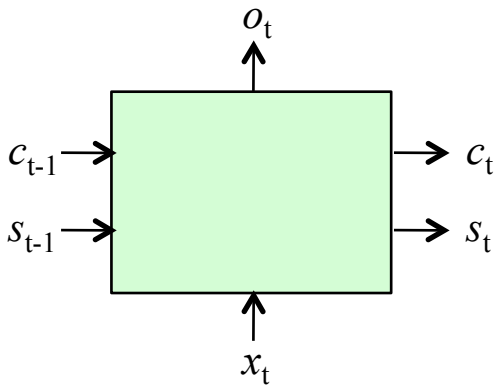


LSTM network

Long Short Term Memory Networks

It contains four layers

1. Forget gate layer f_t to control how much to remember from the previous time steps
2. Input gate layer i_t to control how much to use from the current time step
3. Output gate layer $\tilde{o}_t$
4. Tanh layer $\tilde{c}_t$



$$
\begin{aligned}
f_t &= \sigma(U_f x_t + W_f s_{t-1}) \\
i_t &= \sigma(U_i x_t + W_i s_{t-1}) \\
\tilde{o}_t &= \sigma(U_o x_t + W_o s_{t-1}) \\
\tilde{c}_t &= \tanh(U_c x_t + W_c s_{t-1})
\end{aligned}
$$

sigmoid gives values in between 0 and 1
0: completely forget
1: completely keep

tanh gives values in between -1 and 1

$$
\begin{aligned}
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
s_t &= \tilde{o}_t \circ \tanh(c_t)
\end{aligned}
$$

$\circ$ *is an entry-wise product*